

Les structures de contrôles

Plan

1. *Sommaire*
2. *Conditionnelles*
3. *Répétitions*
4. *Introduction aux listes de compréhension*

Sommaire

- Les structures de contrôle décrivent **l'enchaînement des instructions**. Elles permettent des traitements séquentiels, **conditionnels** ou **répétitifs** (itératifs).
- On dénombre trois structures de contrôles en Python qui permettent d'organiser le code, définies par les instructions :
 - ✓ if
 - ✓ while
 - ✓ for

Chacune de ces structures est de la forme :

instruction condition:

<bloc de lignes>

else:

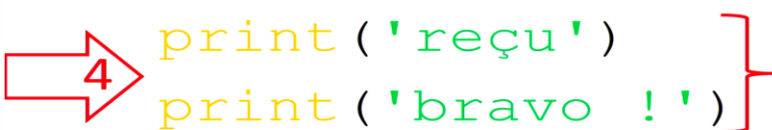
<bloc de lignes>

Conditionnelles

L'instruction if else

Dans cette section , nous allons parler de l'instruction **if else** qui permet de faire de l'exécution conditionnelle. C'est-à-dire qu'un morceau de code va s'exécuter en fonction du fait qu'un **test** soit **vrai** ou **faux**.

```
note = 8
if note > 10:
    print('reçu')
    print('bravo !')
else:
    print('recalé')
```



- **if** est l'*instruction*. **note > 10** est l'expression. **print('reçu')** et **print('bravo !')** forment le **bloc d'instructions**.
- Le **:** est systématique avant un bloc d'instruction. Un bloc d'instructions est un **ensemble d'instructions** qui sont toutes **indentées** du même nombre de caractères vers la droite.
- La convention étant d'indenter tous les blocs d'instructions de **4 caractères** vers la droite. Si le test **if** est **vrai**, Python exécute les instructions du bloc d'instructions.

Conditionnelles

L'instruction **if / elif / else** et opérateurs booléens

Le bloc d'instruction ne sera exécuté que si le test est vrai. Ensuite, on peut On peut ajouter autant de clause **elif** (contraction de **else if**) que de conditions à tester si les précédentes ont renvoyé **False** .

```
if test1:  
    <bloc d'instructions 1>
```

```
elif test2:  
    <bloc d'instructions 2>
```

```
elif test3:  
    <bloc d'instructions 3>
```

```
.....  
else:  
    <bloc d'instructions n>
```

L'intérêt de ces **elif** c'est de faire des tests supplémentaires. Le fonctionnement de cette suite de tests est simple. Il fonctionne comme suit si le **test1** est **True** on exécute le bloc d'instruction correspondant. Si **False** on passe à **test2**, si **False** on passe à **test3**. Et si aucun des tests n'est **True**, on finit dans **else** et on exécute le bloc d'instruction de else.

Donc dans une structure **if / elif / else**, un seul bloc d'instruction sera exécuté.

Conditionnelles

Ce que peut contenir un test

Dans un test d'un **if** ou d'un **elif**, on peut avoir n'importe quelle expression. Le test va appeler la fonction built-in **bool** sur le résultat de l'évaluation de l'expression. Donc on a une expression qui va être exécutée. Elle va produire un objet et on va appeler **bool** sur cet objet.

bool(objet) va appeler :

- soit la méthode **objet.__bool__()** qui est une méthode spéciale sur laquelle nous reviendrons dans les leçons sur les classes. Cette méthode **.__bool__()** va retourner **True** ou **False**.
- soit la méthode **objet.__len__()**. Si la méthode **.__len__()** retourne **0** ce sera **False**, si la méthode **.__len__()** retourne quelque chose d'autre, ce sera **True**. L'intuition derrière ça c'est qu'un **objet vide** est considéré comme **False**. Un objet qui **n'est pas vide** est considéré comme **True**.

Conditionnelles

Exemples d'expressions

Un type built-in :

- sera considéré comme **False** s'il est *False, 0, None* ou *n'importe quel type liste, tuple, dictionnaire chaîne de caractères vides*.
- tout le reste est **True**.

```
L = ["marc", 10 ]      #  
                        # Si la liste L était vide, le print(L) ne s'exécuterait pas.  
if L:                  #  
    print(L)           #
```

Conditionnelles

Une comparaison :

On peut aussi mettre des comparaisons : supérieur >, supérieur ou égal >=, inférieur <, inférieur ou égal <=, égal == ou différent !=.

```
a=1 ; b = 2  
  
if a!=b :  
    print("faux")
```

Le test d'appartenance : Le test d'appartenance in

```
if "a" in "marc":  
    print("ok")
```

*#Si la lettre 'a' est dans la chaîne de caractère 'marc',
#Le bloc d'instruction print('ok') est exécuté.*

Conditionnelles

Un retour de fonction :

On peut utiliser un retour de fonction. C'est-à-dire que l'on va évaluer l'objet retourné par l'appel d'une fonction.

`isdecimal()` va retourner un booléen `True` ou `False`. Ici la fonction renvoie `True`, le bloc d'instruction `print(int(s) + 10)` va donc être exécuté.

```
s = "123"
if s.isdecimal():
    print( int(s) + 10 )
```

Conditionnelles

Les opérateurs de test booléen :

Condition 1	Condition 2	Opérateur	Résultat
True	True	or	True
True	False	or	True
False	True	or	True
False	False	or	False

Condition 1	Condition 2	Opérateur	Résultat
True	True	and	True
True	False	and	False
False	True	and	False
False	False	and	False

```
s = "123"                                # On regarde si "1" est dans notre chaîne de caractère
                                         # et que s est décimal
if "1" in s and s.isdecimal():          # alors le bloc d'instruction print( int(s) + 10 ) est exécuté
    print(int(s)+10)                     #
```

Répétitions

En programmation, on est souvent amené à répéter plusieurs fois une instruction. Incontournables à tout langage de programmation, les boucles vont nous aider à réaliser cette tâche de manière compacte et efficace.

L'instruction **for.....in**

L'instruction **for** permet d'exécuter un bloc de lignes en fonction d'une **séquence**. Elle est de la forme :

for **variable** in **sequence**:

<bloc de lignes>

else:

<bloc de lignes>

Si **sequence** possède **n** éléments, le bloc sera exécuté **n fois**, et variable référencera l'élément **sequence[n-1]** qui sera accessible dans le bloc. Lorsque l'exécution est achevée, un bloc de lignes optionnel présenté par **else** est à son tour exécuté.

```
for i in "Bonjour":  
    print(i)  
else :  
    print("un bloc de code optionel")
```

Répétitions

Utilisation de continue et break

continue : interrompt l'exécution de la boucle pour l'élément en cours et passe à l'élément suivant.

```
# Executer attentivement pour voire comment ca ce passe

for i in range(5):
    if i % 2 :
        continue
    print(i)
else :
    print (i)
```

break : interrompt définitivement l'exécution de la boucle et n'exécute pas l'instruction **else**. Cette instruction est utile lorsque l'on cherche à appliquer un traitement à un et un seul élément d'une liste, ou que cet élément est une condition de sortie.

```
for i in range(5):
    if i == 4:
        print("4 a ete trouve")
        break

    print(" On continue ")
```

Lorsque l'exécution est terminée, le dernier élément de la séquence reste toujours accessible par la variable de boucle

Répétitions

L'instruction while

L'instruction **while** permet d'exécuter un bloc de lignes tant qu'une expression est vérifiée en renvoyant **True**. Lorsque l'expression n'est plus vraie, l'instruction else est **exécutée si elle existe et la boucle s'arrête**. **continue** et **break** peuvent être utilisés de la même manière que pour l'instruction for.

```
i = 0
while i < 4 :
    print(str(i))
    i += 1

else :
    print("end")
```

```
i = 0
while i < 5:
    i +=1
    if i == 2:
        continue
    print(str(i))
```

Introduction aux listes de compréhension

- Les compréhension de liste qui permet de manière extrêmement simple et intuitive d'appliquer une opération à chaque élément d'une liste et éventuellement d'ajouter une **condition de filtre**.
- Supposons que l'on souhaite prendre les logarithmes d'une liste d'entiers $a = [1, 4, 18, 29, 13]$.

Premier Technique : utilisation d'une boucle

```
# penser a importer le module math d'abord
#creation de la liste L
L = [1, 4, 18, 29, 13]

# reation de la liste contenant le logarithme de L
liste = []

for i in L :
    liste.append(math.log(i))
```

Deuxième Technique : utilisation d'une liste de compréhension

```
[math.log(i) for i in L]
```